ロボット制御： 速度フィードバック制御
～組み込みプログラミングによるロボット制御～

1　はじめに

産業機器，家電製品，ロボットなどのモータ制御を必要とする機器は非常に多い。近年のマイクロプロセッサの小型化，高機能化によってそのほとんどがコンピュータ制御であり，多種多様なシステムに高度な制御機能を実装することが可能となってきている。

本実験では，マイクロプロセッサを内蔵した簡単な移動型ロボットを対象として，車軸に内蔵されたエンコーダ信号（車輪角度検出用信号）を利用した移動ロボットの速度フィードバック制御の実装を行う。講義等で習得してきた制御技術を利用する上で大変意味のある実験であり，コンピュータ制御による制御システムの実構成の理解が可能である。実験第 1 日目は，各車輪ごとに回転速度を制御する PI 制御系を実装して，目標速度への応答性能を評価する実験を行う。実験 2 日目は，1 日目の車輪毎の制御系を発展させて，移動ロボットの運動ベクトル（並進，回転運動）を直接制御できるよう改良を行う。これにより，任意の軌道を描くよう動作させることが可能となり，実験では円周に沿った軌道などをたどるような動作をさせて，その制御効果の確認を行ってみる。

2　実験に利用する移動ロボットのハードウェア構成

ロボット (図 1) は教育玩具用に市販されているものであるが，組み込み装置向けに広く普及しているルネサスエレクトロニクス社製の H8-Tiny シリーズのマイクロプロセッサ（H8/3672 型番：HD64F3672FP）を搭載している。本プロセッサは，16 ビット高速 CPU であり (動作周波数 16MHz)，ROM（フラッシュメモリ）を 16KB, RAM を 2048 バイト（2KB）を内蔵しているため外付けのメモリを必要とせず，ワンチップ動作が可能である。また AD 変換機能を内蔵しているため，アナログ出力のセンサ類を直接接続することができる。またデジタル信号の入出力が可能であり，エンコーダ信号，スイッチ類や LED などの表示装置と簡単に接続できる。すなわち，図 1(c) に示すように今回のロボットのほぼ全機能がワンチップで管理可能となっている。(詳細な回路図は章末付録 A を参照)

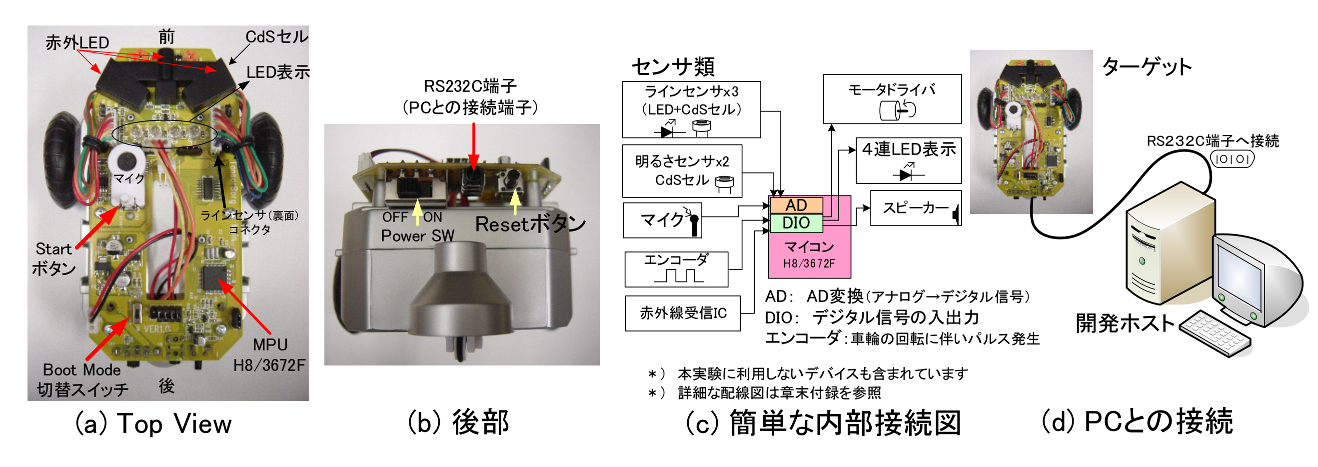


図 1: 実験に利用する移動型ロボットと接続構成

3　モータの位置や速度を制御するための一般的な構成

3.1　エンコーダ信号による位置検出

モータの位置や速度を目標値に制御するためには，センサで現在の位置や速度を観測し，フィードバック制御系を構成するのが一般的である。工作機械や産業用ロボットなどの精度を要求する機器のモータ制御においては，**エンコーダ**と言われる位置検出用のデジタルコードを生成する機構デバイスを回転軸等に備えて位置を検出することが多い。速度制御であれば，タコメータに代表されるアナログ速度検出センサを回転軸に取りつけることもあるが，エンコーダによる位置検出結果をコンピュータで微分処理して速度信号に変換し，フィードバック制御系を構成することが多くなっている。

図 2 に一般的に多く利用されている光学式エンコーダ装置の構成を示している。LED 光が回転円盤に刻まれたスリットを通して反対側の受光素子に照射されている。軸が回転すると，受光素子の出力電圧がパルス状に変化することになる。出力されるデジタル符号のパターンは，インクリメンタル型，アブソリュート型の 2 通りに分けられるが，図 2 はインクリメンタル型の例である。この場合，回転に応じてパルスが出力され，そのパルスを数えるカウンタ回路やソフトウェアを構成すれば，回転軸の角度を検出できることになる。図の中では説明簡単化のため内と外で位相をずらして 2 つのスリット列があるエンコーダ装置を例示しているが，基本的に位相差を与えることで回転方向の検出が可能となる（位相差の生成には，受光素子の配置を工夫することで 1 つのスリット列で構成することが一般的である）．アブソリュート型エンコーダ装置の場合，回転軸角度の絶対値に相当する符号が 2 進符号などで直接出力されるようスリットを構成し，近年普及が進んでいる。

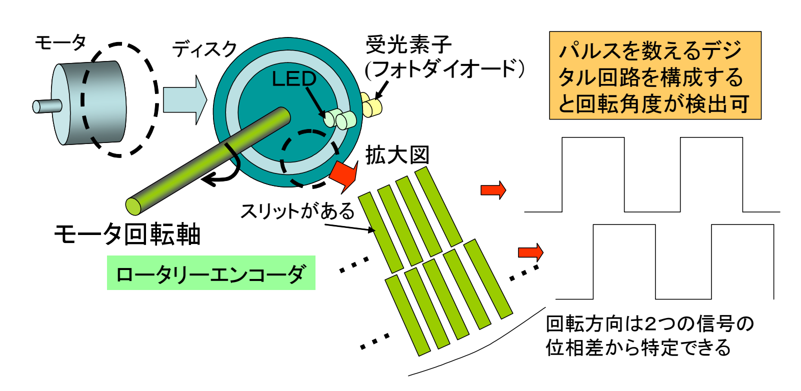


図 2: 一般的な光学式エンコーダの構成

今回利用するロボットの車輪回転軸には，インクリメンタル型のとても簡単な仕組みのエンコーダがギアボックスの中に組み込まれている。図 3 に示すように，車輪伝達軸の一つに白黒の帯を側面に描いた回転円盤があり，LED 光を白黒の帯の回転面に照射し，その反射光を受光素子で受ける構成となっている。白面と黒面で受光量が変化するため，車輪が回転すると受光素子の信号がパルス状に変化することになる。このパルスを数えれば車輪の回転角度が検出できることになる。ただし，車輪が逆方向に回転しても同様のパルスが生じるために，正転逆転の区別ができず，少々制約のある構成である。

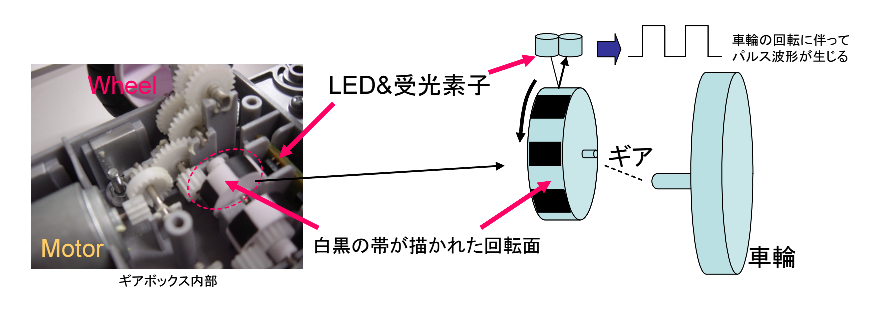


図 3: 実験で利用するロボットのエンコーダ信号

3.2　PI 制御による速度制御系の構成

検出されたエンコーダのパルス数をロボットの車輪回転角に換算すると，ロボットの位置や速度のフィードバック制御系が構成できる。今回の実験では，正転逆転の区別ができないため，位置制御系を構成するにはちょっと工夫が必要なため，速度フィードバック制御系を実装してみることとする。

単なる比例制御（P 制御）での制御ブロック図は図 4 のような構成例となる（今回のロボットは厳密にはモータ電流を制御する駆動回路ではない）。モータ部分とモータ駆動回路部分を除いたほとんどの部分がマイクロコンピュータで処理される。実際の速度はエンコーダからのパルス数から角度に換算し，その微分量が実際の速度として用いられる。

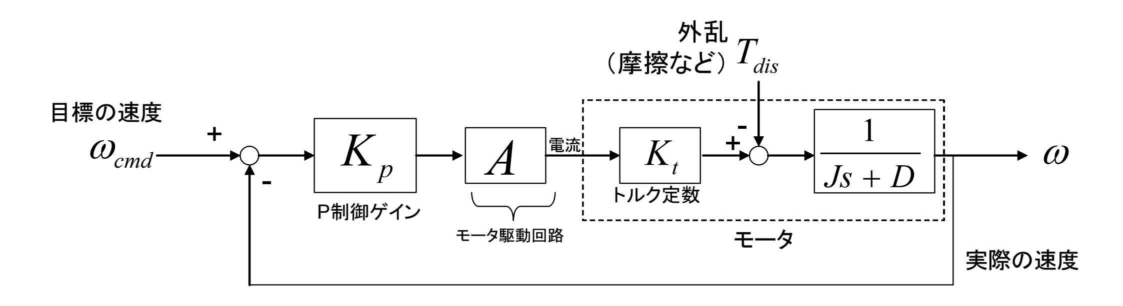


図 4: P 制御のブロック図

今回実験に利用するロボットでは，特にギアボックスでのトルク伝達機構の中に比較的大きな摩擦の影響が存在する。P 制御のみでは，目標とする回転速度の精度の良い応答を得ることは大変難しい。このことは 1 日目の実験検証でも確認できる事項である。そこで制御誤差を効果的に抑圧する手法として積分制御（I 制御）を追加した PI 制御系を図 5 のように構成する。これにより比較的精度の良い速度制御系の実装が行える。

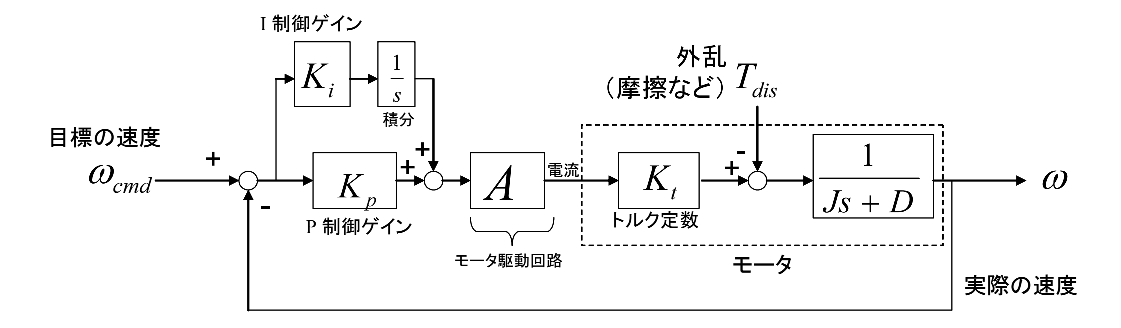


図 5: PI 制御のブロック図

4　第 1 日目の実験課題

4.1　車輪回転パルス数の確認実験

【1】C 言語ソースコードなどのテンプレートファイルをコピー

各 PC のデスクトップに「(秋) ロボット制御実験用テンプレート」フォルダが置いてあるので，同じくデスクトップの「光システム実験 2」の中の午前グループは「AM」，午後グループは「PM」フォルダを開き，そこにコピーする。そして，フォルダの名前を「ロボット制御実験用テンプレート」から，実験した日付「X 月 Y 日」に変更する。

【2】統合開発環境 HEW4 の起動

コピーしてきたフォルダを開いて,「RobotJikken.hws」をダブルクリックして起動する。そのファイルには開発するターゲット MPU の情報やコンパイル環境，及びメモリ空間の利用設定など必要な設定が保存してあり，その設定が読み込まれた上で HEW4（図 6）が起動する。(フォルダが移動されましたとの主旨のウィンドウが出たら，そのまま OK とする）

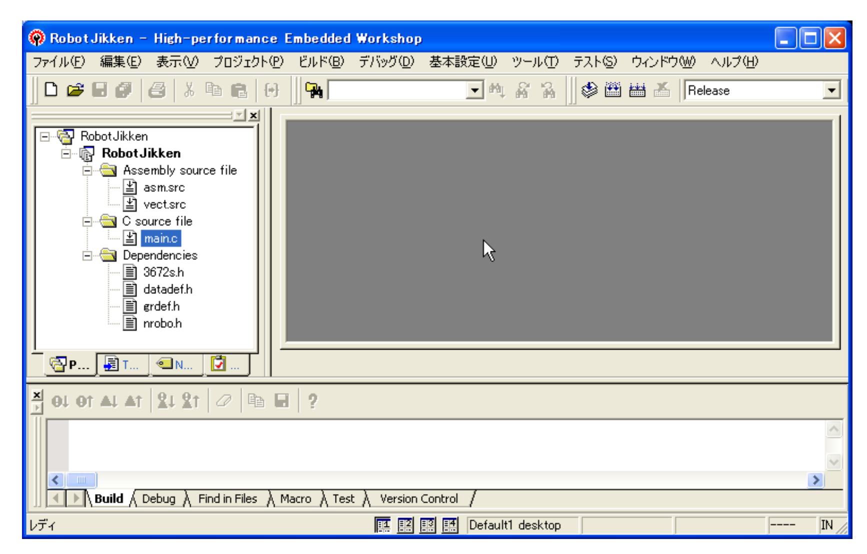


図 6: HEW4 の起動画面

左上側の枠にテンプレートのプロジェクトで利用しているファイルがツリー形式で表示されている。プログラムの起動を担う部分のみアセンブリ言語で記述してあり"Assembly source file" に入っている。メインとなるプログラムは"C source file" に分類されている「main.c」である。main.c をダブルクリックすると，右の枠内に C 言語ソースが表示される。

【3】プログラムの記述

main 関数の冒頭の初期設定の部分は削除しないこと。各種レジスタの初期化，グローバル変数の初期化などを行っている。`while(1)` の部分から無限ループがはじまり，その後の `while(1)` ··· の部分でロボット上の Start ボタンが押されるまで無限に待つための記述 1 行がある。今回の実験で記述する部分は，`user_main()` 関数の部分のみであり，main 関数から呼び出されている。しがたって，Start ボタンが押されると実行がはじまる。また，`user_main()` 関数が `#pragma` ··· で囲まれているが，これはその関数の実行コードは RAM 領域に置くことを意味している。

テンプレートファイルでは，`user_main()` 関数の中身を記述していないので，次に示すコードを追加しなさい。エンコーダ信号のパルスは内部の割込処理で常にカウントされているため，ここでの処理は右車輪のカウント値 `wheel_cnt_right`，左車輪の値 `wheel_cnt_left` をゼロに初期化して，その値を PC に表示するために送信処理をしているのみである。これら 2 つの変数は int 型のグローバル変数として宣言済である。`user_main()` 関数内で宣言する必要はない。

各車輪のカウント値を確認するコード

```
#pragma section USR
void user_main(void)
{
        float t=0.0;

        wheel_cnt_right=0;
        wheel_cnt_left=0;

        while(1){
                Wait( 100 );//1s の待ち

                //カウント値を RS-232C 経由で PC に送信
                print_float( t );              //時間 t を送信
                print_string( "\t" );          //タブを送信
                print_int(wheel_cnt_right);    //右車輪カウント値を送信
                print_string( "\t" );          //タブを送信
                print_int( wheel_cnt_left);    //左車輪カウント値を送信
                print_string( "\r\n" );        //改行コードを送信

                t=t+1.0;
        }
}
#pragma section
```

【4】ロボットに転送して，実行してみる

プログラムをビルドした後，PC 画面デスクトップ上の SysJikken.exe を起動して，ロボットの RAM へプログラムを転送する（ROM には既に必要なプログラムが書き込まれている。RAM への転送手順を忘れた場合は付録 C を参照）。

SysJikken.exe 上の「受信開始」ボタンを押して，ロボットから送信されてくる文字列を受け取れるようにしておく。その後，ロボット上のスタートボタンを押して，動作を開始させる。SysJikken.exe 右端の受信端末欄に経過時間，右車輪カウント値，左車輪カウント値が表示されるか確認する。

カウント値は最初はゼロがつづけて表示されているはずである。**注意深くロボットの車輪を手でゆっくり回転させてみて**, カウント値の変化をみてみなさい。車輪を１回転させたとき，カウント値はいくらになるか。このロボットでは車輪 1 回転で１２０パルスとなるはずである。左右の車輪とも正常にカウントできているか確認しなさい。ただし，正転逆転ともにカウントされるので，一方向に回転させること。

（確認が終わったら,「受信開始」ボタンを再度押して，受信状態を解除しておくこと。）

4.2 P 制御による速度フィードバック制御

左右両方の車輪とも P 制御系を構成して，ロボットの前進速度を制御する系を構成してみる。

【1】カウント値を車輪の回転角度に換算

先の実験で表示されたカウント値を車輪の角度（ラジアン単位）に換算する。車輪 1 回転で１２０パルスであるから，次の式で換算できる。

$$\theta_{right} = \frac{\texttt{wheel_cnt_right}}{120} \times 2\pi \tag{1}$$

次のコードを参考にして，角度を PC に表示するプログラムに修正して，先ほどと同様に実行して確認してみなさい。なお，π=3.14 でよい。車輪 1 回転で 2π となるか見てみるとよい。

★追加／修正した行の行頭には，―>印 を入れている

各車輪の角度を確認するコード

```
#pragma section USR
void user_main(void)
{
        float t=0.0;
->      float theta_r, theta_l;   //← 角度の変数宣言を追加

        wheel_cnt_right=0;
        wheel_cnt_left=0;

        while(1){
             Wait( 100 );//1s の待ち
             //角度の換算
->           theta_r = (float)wheel_cnt_right/120.0 * 2.0* 3.14  ;
->           theta_l = (float)wheel_cnt_left /120.0 * 2.0* 3.14  ;

             //カウント値を RS-232C 経由で PC に送信
             print_float( t );
             print_string( "\t" );
->           print_float(theta_r);   //← 角度の送信に変更 ‥_float()に変更
             print_string( "\t" );
->           print_float( theta_l);  //← 角度の送信に変更 ‥_float()に変更
             print_string( "\r\n" );

             t=t+1.0;
        }
}
#pragma section
```

【2】回転角度を微分して，回転速度を算出する

得られた回転角度を微分演算することで回転速度 (rad/s) を得ることができる。回転速度を得るコードは次頁に示しているが，以下の説明はそのコードの解説である。

現時点の速度を得るには，１サンプリング時間前（1 回前のループ）からの角度の変化量をサンプリング時間 (1 ループ当りの時間) で割れば良い。したがって，右車輪の回転速度 ω_{right} は，

$$\omega_{right}(k) = \frac{\theta_{right}(k) - \theta_{right}(k-1)}{\Delta t} \tag{2}$$

となる（左車輪も同様である）。$\theta_{right}(k)$ は現在の角度，$\theta_{right}(k-1)$ は１サンプリング時間前の角度である。その差をサンプリング時間で割っていることになる。

しかしながら，今回のロボットのエンコーダ信号の分解能（1 回転あたりのパルス数）が十分とは言えないため，(2) 式で得られる速度信号には微分ノイズ（信号のばらつき）が多く含まれる。そこで (2) 式の信号に対して簡単なローパスフィルタ（低域周波数成分のみ取り出すフィルタ，すなわち信号を平滑化する）を適用する。今回は以下の簡単なデジタルフィルタで構成する。

$$\omega'_{right}(k) = \alpha\ \omega_{right}(k) + (1-\alpha)\ \omega'_{right}(k-1) \tag{3}$$

この (3) 式で得られた速度をフィードバック信号として利用することになる。$\omega_{right}(k)$ は現時点で算出された速度，$\omega'_{right}(k-1)$ は１サンプリング時間前の速度（フィルタ後）である。また，上式の α は０〜１の範囲で定める定数であるが，今回はそれほど早い速度変化の制御を必要としないので，$\alpha = 0.1$ 程度で良しとする。

★追加／修正した行の行頭には —＞印を入れている

各車輪の回転速度を算出する部分まで追加したコード

```
#pragma section USR
void user_main(void)
{
        float t=0.0;
        float theta_r, theta_l;
->      float theta_old_r=0.0, theta_old_l=0.0;
->      float omega_r, omega_l;
->      float omega_old_r=0.0, omega_old_l=0.0;
->      float alpha=0.1;
->      int   loop_counter=0;

        wheel_cnt_right=0;
        wheel_cnt_left=0;

        while(1){
->           wait_for_10ms_timer();   //← サンプリング時間をきっちり 10ms とする
             //角度の換算
             theta_r = (float)wheel_cnt_right/120.0 * 2.0* 3.14  ;
             theta_l = (float)wheel_cnt_left /120.0 * 2.0* 3.14  ;
             //速度の算出
->           omega_r= (theta_r-theta_old_r)/0.01; //(2) 式参照
->           omega_l= (theta_l-theta_old_l)/0.01;
->           omega_r= alpha * omega_r +(1-alpha)*omega_old_r;//(3) 式参照
->           omega_l= alpha * omega_l +(1-alpha)*omega_old_l;

->           theta_old_r=theta_r;// ←次回のループでθ(k-1) として利用
->           theta_old_l=theta_l;
->           omega_old_r=omega_r;// ←次回のループでω'(k-1) として利用
->           omega_old_l=omega_l;


             //
             // このあたりに P 制御や PI 制御の処理を後ほど記述する
             //


             //カウント値を RS-232C 経由で PC に送信
->           if( (loop_counter%10)==0 ){   //← 10 回に 1 回のみ送信するように変更
               print_float( t );
               print_string( "\t" );
->             print_float( omega_r);   //← 速度の送信に変更
               print_string( "\t" );
->             print_float( omega_l);   //← 速度の送信に変更
               print_string( "\r\n" );
->           }
->           loop_counter++;

->           t=t + 0.01;            //← 1 回のループ時間＝ 0.01 秒
        }
}
#pragma section
```

プログラムをビルドしてロボットに転送し，実行してみなさい。先ほどと同様にして車輪を手で動かしてみて，受信端末に車輪の回転速度らしい数値が表示されるか確認しなさい。表示される速度の数値の単位は rad/s である。

【3】**P 制御によるフィードバック部分を追加**

得られた速度信号をフィードバック信号として P 制御を行う。

$$u_{right} = K_p(\omega_{cmd} - \omega_{right}) \tag{4}$$

ここで，K_p は比例制御ゲイン，ω_{cmd} は車輪の目標回転速度 (rad/s)，ω_{right} は現在の車輪の回転速度である。算出された u_{right} をそのままモータの駆動用信号として利用する (左車輪も同様)。すなわち，

```
Motor_R( (int)u_right );
Motor_L( (int)u_left );
```

のようにモータパワーを指定できる Motor_R()/Motor_L() 関数を利用してモータを駆動する（関数は章末付録 B を参照)。適宜，変数を用意して，P 制御の記述を先のプログラムの「このあたりに ⋯ 」部分に追加して試してみなさい。とりあえず，以下の制御パラメータで良い。

$$K_p = 2.0, \quad \omega_{cmd} = 2\pi\ [rad/s]$$

【4】**応答結果をグラフで確認**

SysJikken.exe の受信端末欄下に「グラフ表示」ボタンがある。クリックするとグラフが表示され，端末上に表示される数値をリアルタイムに図 7 のようにグラフ化してくれる（横軸：時間，縦軸：速度)。「受信開始」ボタンを押してデータを受け取れる状態にしておいて，ロボットのスタートボタンを押して動作を開始する。すると，受信端末欄に数値が表示されるとともに，グラフ上にもプロットされるようになる。ロボットの速度応答の様子をグラフで確認しなさい。そして以下の空欄を埋めなさい。

✎ 以下の K_p の値における応答の様子を記録しなさい

K_p の値	応答の収束の速さは	オーバーシュートがあるか	収束した速度 (rad/s)
1.0			
2.0			
4.0			
6.0			



図 7: 受信データのグラフを表示する機能

4.3 PI 制御による速度フィードバック制御

先の実験の P 制御では，比例制御ゲインを上げることで目標速度に近づけることができるが，それに伴いオーバーシュート（行き過ぎ）が観測される傾向がある。ギアボックス内の比較的大きな摩擦に打ち勝つため

にゲインを上げたいところであるが，モータに大きなパワーを要求することになり，ステップ状の指令に対してモータの出力限界（出力飽和）を越えてしまう。それにより，過渡応答（応答の立ち上がり）時において意図した制御効果が働かず，応答のオーバーシュートを引き起こすと考えられる。

制御誤差を効果的に抑圧する方法である積分制御（I 制御）を加えることで解決できる。比例制御ゲインを増やすことに依存せず，効果的に摩擦等の外乱（制御を妨げる要因）を抑えることができる。P 制御と I 制御を組み合わせた制御を PI 制御という。（これに微分制御も加えたものを PID 制御というが，今回は触れない。）

【1】PI 制御となるよう記述を追加修正する

制御誤差 e_{right} は次のように目標値と現在速度の差で計算できる。

$$e_{right} = \omega_{cmd} - \omega_{right} \tag{5}$$

比例制御と積分制御を合わせると，以下の式でモータを駆動することになる。なお，s はラプラス演算子であり，$1/s$ は積分を意味することになる。

$$u_{right} = \underbrace{K_p\, e_{right}}_{P\text{ 制御}} + \underbrace{K_i\, \frac{e_{right}}{s}}_{I\text{ 制御}} \tag{6}$$

さて，この演算をプログラムで表現する必要があるが，右辺第 1 項の P 制御の部分は単なる乗算である。第 2 項の積分計算の部分について，どのように記述するか以下に解説する。

第 2 項の計算の意味は，制御誤差 e_{right} を積分して，それに積分制御ゲイン K_i を掛け合わせることになる。コンピュータによる数値積分の手法は数多く存在するが，図 8 に示すように値の変化を長方形で近似した最も簡便な方法を採用することとする。

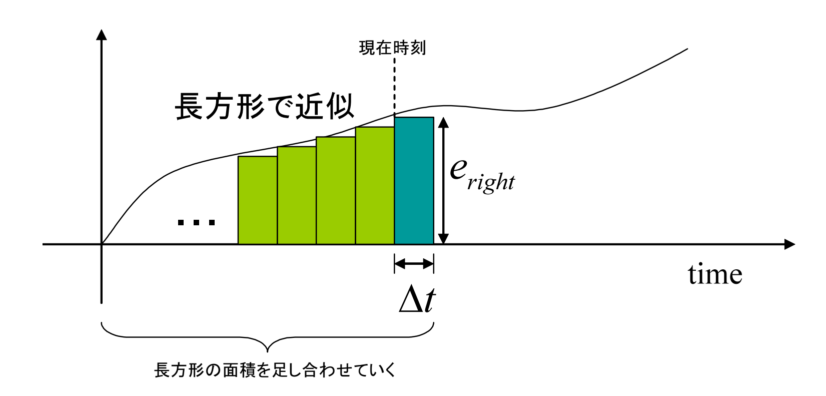


図 8: 数値積分の様子

基本的に積分は図 8 塗り潰し部分の面積であるから，サンプリング毎に長方形の面積を足し合わせていけば良い。すなわち，プログラム風に式を書けば，次のようになる。

$$\texttt{u_integ_right} = \texttt{u_integ_right} + K_i * e_{right} * \Delta t \tag{7}$$

`u_integ_right` が (6) 式右辺第 2 項の計算値であるが，この変数の初期値をゼロとしておくことを忘れないようにする。Δt はサンプリング時間であるが，この場合は 0.01 秒である。

適宜，変数を用意して，P 制御の記述を PI 制御に変更し，動作させてみなさい。とりあえず，以下の制御パラメータで良い。

$$K_p = 2.0,\ \ K_i = 5.0\ \ \omega_{cmd} = 2\pi\ [rad/s]$$

【2】応答結果をグラフで確認

先ほどと同様にして，ロボットの速度応答の様子をグラフで確認しなさい。そして以下を埋めなさい。

✎ $K_p = 2.0$ に固定，K_i の値をいろいろ変えて，応答の様子を記録しなさい

K_i の値	応答の収束の速さは	オーバーシュートがあるか	収束した速度 (rad/s)
0.0			
2.0			
5.0			
8.0			

【3】P 制御と PI 制御の応答結果を印刷

P 制御と PI 制御の比較結果を印刷するために，右車輪の制御を P 制御（単に右車輪のゲイン K_i のところをゼロにすれば良い。K_p=2.0），左車輪の制御を PI 制御（K_p=2.0, K_i=5.0）として，応答結果を取りなさい（グラフ表示で応答をみながら何度か試してみなさい）。

SysJikken.exe 上の受信端末欄に表示された数値列をエクセルにコピー＆貼付けする。受信端末欄の上で右クリックして「すべてを選択」，その後，同様に「コピー」を選択。エクセルを起動後，最左上のセルから貼り付ければ良い。貼り付け後，そのまま「グラフ」アイコンボタンをクリックして，グラフ化する。座標軸タイトルや単位等もきちんと入力してから印刷する。

4.4 滑らかに発進／停止するようロボットを速度制御する

最後に，ここまでに作成した制御プログラムを応用して，等加速，等速走行，減速するよう速度指令を与えて，滑らかに発進／停止するような前進走行の PI 制御を行ってみる。

図 9 に示すような車輪の回転速度指令を与えるよう，プログラムを修正する。最初の 2 秒間は 3 rad/s^2 の加速，その後の 2 秒は 6 rad/s の等速運動，最後の 2 秒で $-3rad/s^2$ の減速で停止するような速度軌道である。

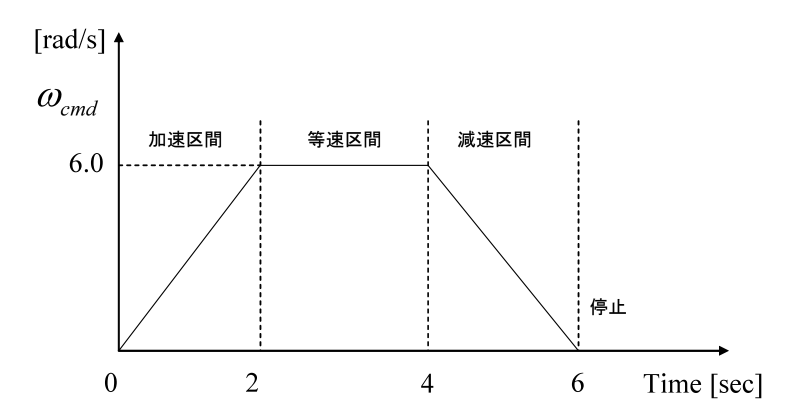


図 9: 加速／等速／減速による滑らかな走行指令

【1】速度指令を与える記述を追加

時間とともに指令速度の式を切り替える必要がある。参考プログラムを以降に示しているので参照すると良い。図 9 のグラフのように目標速度が与えられるよう，適切な式を入れるのみである。

【2】速度の応答結果をエクセルから印刷

グラフ表示をさせながら，速度応答の追従性を確認してみなさい。良い応答が得られたら，先ほどと同じ手順で応答結果をコピー＆貼り付けでエクセルにコピーし，グラフを印刷しなさい。

✎ エンコーダ信号が車輪の正転逆転に対応していないため，速度をゼロに維持する制御が困難である。そのため，減速して停止した際に暴走することがあるので注意すること。（速度指令がゼロになったと同時にモータのパワー指令をゼロにすれば解決できるが，今回は特に対策しなくても良しとする）

【3】プログラムの印刷

✎ **注意）main.c を全部印刷しないこと！量が多いため**

印刷は user_main() 関数の部分のみで良い。user_main() 関数の部分をマウスで選択した状態にして，ファイルメニューの印刷を実行する。印刷範囲が「選択した部分」となっていることを確認後，「OK」ボタンをクリックする。

加減速指令を加えるための追加コード

```
#pragma section USR
void user_main(void)
{
                .
                .
                .
                //
                // このあたりに P 制御や PI 制御の処理を後ほど記述する
                //
                if( t < 2 ){
                    //加速区間の速度指令
                    omega_cmd = __________ ;
                }else if( t < 4 ){
                    //等速区間の速度指令
                    omega_cmd = __________ ;
                }else if( t < 6 ){
                    //減速区間の速度指令
                    omega_cmd = __________ ;
                }else{
                    //最後に停止
                    omega_cmd = 0 ;
                }

                //PI 制御の記述
                .
                .
          }
}
#pragma section
```

5　第 2 日目の実験課題

5.1　はじめに

第 1 日目の実験では，各車輪の回転速度を制御する系を構成した。しかし，実際上は想定した軌道に沿ってロボットを走行させたいであろうから，そのために逐一左右の車輪にどのような回転速度を与えればいいか指定するのでは少々面倒である。

今回用いている移動型ロボットの場合，並進速度，回転速度でロボットの運動ベクトルを表現することが一般的である。図 10 に示すように車輪の方向である進行方向の並進速度 v（単位は m/s）とロボット姿勢の回転速度 ω（単位は rad/s）の 2 つの速度で表現される。

第 2 日目の実験は，1 日目の制御系をこの並進速度，回転速度を直接制御する制御系に拡張を行う。これにより任意の曲線軌道などへの追従制御が簡便となる。最終的に，円軌道を描いて動作させたり，直進動作と円軌道を組み合わせて指定した軌道を描くようプログラムを作成する。

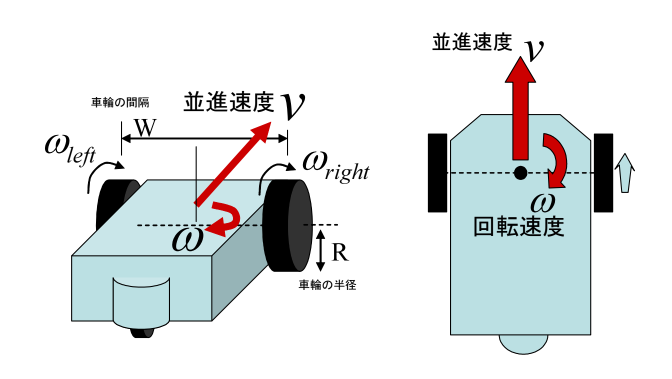


図 10: 移動ロボットの運動ベクトルの定義

5.2　座標変換の概略，及び実験手順

2 つの車輪の回転速度を $[\omega_{right}, \omega_{left}]$ のようにベクトル表現すると，(8) 式の行列ベクトル演算で移動ロボットの並進・回転ベクトル $[v, \omega]$ を得ることができる。運動表現のための座標系を変えているため，座標変換を行っていることになる。

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{W} & -\frac{R}{W} \end{bmatrix} \begin{bmatrix} \omega_{right} \\ \omega_{left} \end{bmatrix} \tag{8}$$

ここで，R は車輪の半径，W は 2 つの車輪の間隔である（図 10 左図参照）。今回の実験では，ここで得られる v, ω に対して PI 制御系を構成する。まず，並進／回転それぞれの制御誤差 e_v，e_ω を求める。

$$\begin{aligned} e_v &= v_{cmd} - v \\ e_\omega &= \omega_{cmd} - \omega \end{aligned} \tag{9}$$

ここで，v_{cmd}，ω_{cmd} は並進／回転速度の目標速度である。次に PI 制御系の式を計算する。

$$\begin{aligned} u_v &= K_p\, e_v + K_i\, \frac{e_v}{s} \\ u_\omega &= K_p\, e_\omega + K_i\, \frac{e_\omega}{s} \end{aligned} \tag{10}$$

さて，ここで得られた制御入力 u_v，u_ω を今度は逆に車輪の回転系の座標表現に戻す必要がある。(8) 式を車輪の回転速度ベクトル $[\omega_{right}, \omega_{left}]$ を求める逆変換の式に書き直すと，次の式が得られる。

$$\begin{bmatrix} \omega_{right} \\ \omega_{left} \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{W} & -\frac{R}{W} \end{bmatrix}^{-1} \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{1}{R} & \frac{W}{2R} \\ \frac{1}{R} & -\frac{W}{2R} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \quad (11)$$

制御入力 u_v，u_ω は，速度の制御誤差を減らすための加速度の次元であるが，(11) 式の行列部分は定数行列なので座標系間の加速度関係も (11) 式と同様の行列の積算で変換できる。したがって，次式の演算で制御入力 u_v，u_ω を左右車輪のモータの制御入力として換算することができる。

$$\begin{bmatrix} u_{right} \\ u_{left} \end{bmatrix} = \begin{bmatrix} \frac{1}{R} & \frac{W}{2R} \\ \frac{1}{R} & -\frac{W}{2R} \end{bmatrix} \begin{bmatrix} u_v \\ u_\omega \end{bmatrix} \quad (12)$$

最終的に，得られた u_{right}，u_{left} を使って，各モータのパワーを指定する関数によりモータを駆動すれば良い。

```
Motor_R( (int)u_right );
Motor_L( (int)u_left );
```

制御ブロック図で表現すると，図 11 となる。図中の行列 T は (8) 式中の行列である。

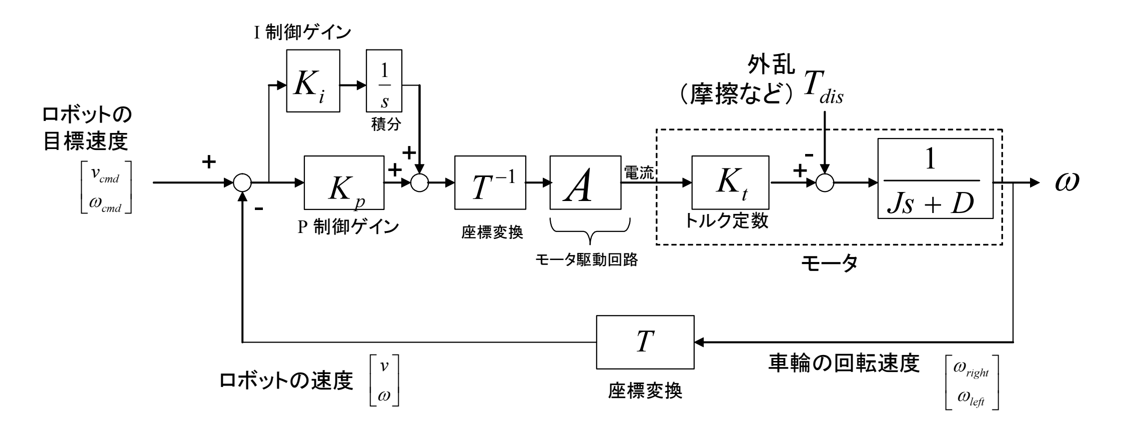


図 11: ロボットの並進／回転速度を制御する PI 制御のブロック図

【1】1 日目のプログラムのフォルダを丸ごとコピー

1 日目のフォルダを丸ごとコピーして今日の日付に変更する。フォルダを開いて,「RobotJikken.hws」をダブルクリックして統合開発環境 HEW4 を起動する。

【2】座標変換を実装したプログラムに変更する

プログラムのアウトラインを以降に示しているので参考にすると良い。制御系を構成している座標空間が変更されたために，制御ゲインは 1 日目から少し調整が必要となる。とりあえず以下のパラメータで動作の確認を行いなさい。目標速度の回転速度をゼロとしているが，つまり前進方向に制御されるはずである（ロボット姿勢角の変化なし）。また，車輪半径 R や車輪間隔 W はメジャーで実測して与えなさい (単位は m で与えること)。

$$K_p = 6.0,\ K_i = 12.0,\ v_{cmd} = 0.2\ [m/s],\ \omega_{cmd} = 0.0\ [rad/s]$$

座標変換を実装したプログラムコードのアウトライン

```
#pragma section USR
void user_main(void)
{
        float t=0.0;
        float theta_r, theta_l;
        float theta_old_r=0.0, theta_old_l=0.0;
        float omega_r, omega_l;
        float omega_old_r=0.0, omega_old_l=0.0;
        float alpha=0.1;
        int   loop_counter=0;
->      // 必要な変数を適宜追加すること，変数の名前は自分で適当に決めなさい
->
        wheel_cnt_right=0;
        wheel_cnt_left=0;
        while(1){
             wait_for_10ms_timer();   //← サンプリング時間をきっちり 10ms とする
             //角度の換算
             theta_r = (float)wheel_cnt_right/120.0 * 2.0* 3.14  ;
             theta_l = (float)wheel_cnt_left /120.0 * 2.0* 3.14  ;
             //速度の算出
             omega_r= (theta_r-theta_old_r)/0.01; //(2) 式参照
             omega_l= (theta_l-theta_old_l)/0.01;
             omega_r= alpha * omega_r +(1-alpha)*omega_old_r;//(3) 式参照
             omega_l= alpha * omega_l +(1-alpha)*omega_old_l;

             theta_old_r=theta_r;// ←次回のループでθ(k-1) として利用
             theta_old_l=theta_l;
             omega_old_r=omega_r;// ←次回のループでω'(k-1) として利用
             omega_old_l=omega_l;

->           // (8) 式の座標変換で，v， ω を算出

->           // (9) 式で，制御誤差 e_v， e_ωを得ておく

->           // (10) 式で，PI 制御系を記述  (1 日目のプログラムを参考にする)

->           // (12) 式で，各車輪の座標表現に戻す

             Motor_R( (int)u_right );
             Motor_L( (int)u_left );

             //カウント値を RS-232C 経由で PC に送信
             if( (loop_counter%10)==0 ){
               print_float( t );
               print_string( "\t" );
               print_float( omega_r);
               print_string( "\t" );
               print_float( omega_l);
               print_string( "\r\n" );
             }
             loop_counter++;

             t=t + 0.01;            //← 1 回のループ時間＝ 0.01 秒
        }
}
#pragma section
```

【3】円周に沿って運動するよう動作させてみる

並進速度の目標値 v_{cmd} とロボット姿勢の回転速度 ω_{cmd} を適切に与えると，いろいろな軌道を描いて動かすことができる。最も単純な曲線として円軌道を試してみる。等速円運動は一定の並進速度に加えて，一定の回転運動が同時に起きている運動である。直径 0.25m の円周を 4 秒で周回するような動作をさせてみなさい。スタート後，延々と周回しつづける動作で良い。

✎ 円 1 周の長さを 4 秒間で等速運動するから，並進速度（進行速度）が算出できる。回転運動は，ロボットの姿勢角（向き）は 1 周で 360 度の回転が必要であり，それに 4 秒必要であるから，回転速度（rad/s の単位で与えること）も決まる。

【4】以下の動作を実現してみなさい

以下の図 12 の (A) あるいは (B) の軌道を描くようプログラムを変更しなさい（ (B) は少々面倒なので自信のない班はとりえあえず (A) に取り組むと良い)。なお，スタート後，延々と周回しつづける動作で良い。追加する個所は，(8) 式の記述の前あたりで良い。

✎ 1 周の秒数が指定されているから，1 周の走行距離を算出すると，等速での並進速度が算出できる。するとカーブの際に必要なロボット姿勢の回転速度も算出できる。あとは，動作を切り替える地点の時間を算出すれば，1 日目の課題「滑らかな発進／停止 … 」のところで if 文を使って記述したように，時間とともに速度指令を切り替えればよい。

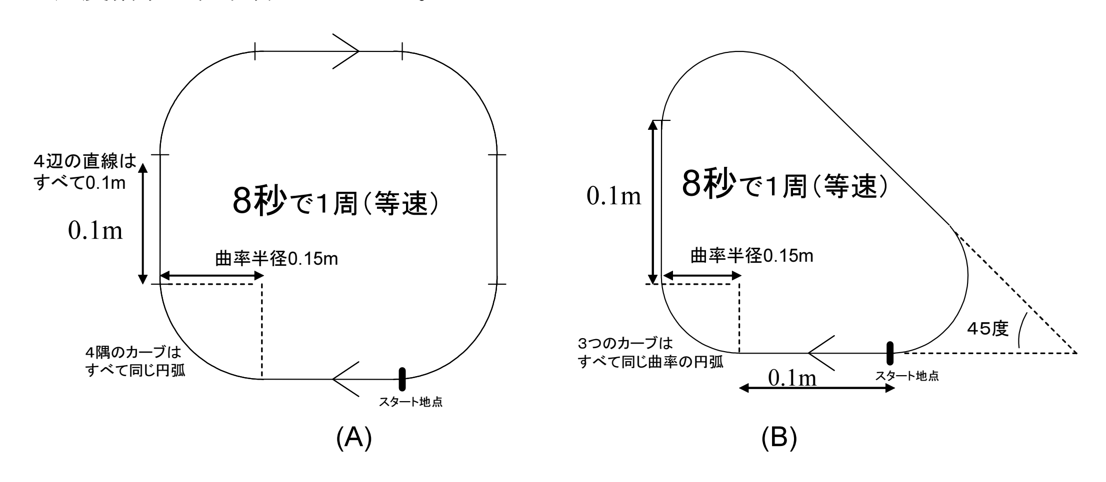


図 12: 計画軌道

【5】プログラムの印刷

✎ <u>**注意）main.c を全部印刷しないこと！量が多いため**</u>

印刷は `user_main()` 関数の部分のみで良い。`user_main()` 関数の部分をマウスで選択した状態にして，ファイルメニューの印刷を実行する。印刷範囲が「選択した部分」となっていることを確認後，「OK」ボタンをクリックする。

6　レポートについて

レポートには作成したプログラム（第 1 日目，2 日目）と 1 日目の応答結果（グラフ）を添付してもらうので，実験終了後，作成したプログラムやグラフを印刷するのを忘れないようにする。レポートには実験の目的，及び第 1 日目，2 日目の実験内容について，簡潔に書くこと。

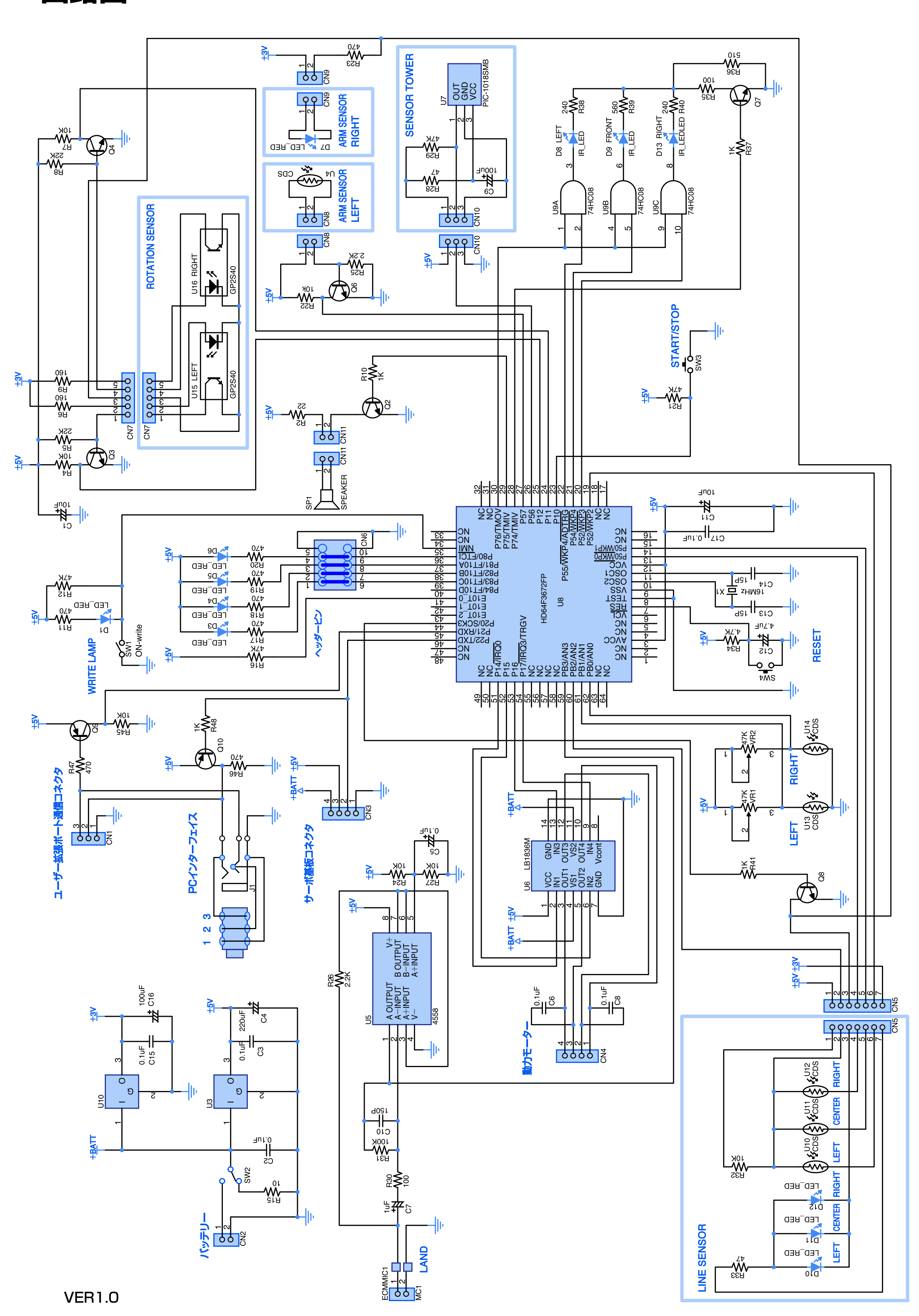
実験内容に関する考察では，第 1 日目：印刷したグラフの応答結果（P 制御と PI 制御の比較，加減速制御の結果）に対する考察をすること。また，PI 制御で速度を制御する上で，K_p と K_i の選定方法を考察し，整理してまとめなさい。今回は摩擦が主な制御誤差の要因となっていたが，その他に制御外乱（制御を乱す要因や制御誤差の要因）として考えられるものを挙げてみなさい。第 2 日目：与えた軌道を長く動かしていると軌道がずれてくるが，その原因について考察しなさい。またそのずれを小さくするためにはどうしたら良いか考えてみなさい。

最後にまとめを書くことを忘れないようにする。実験目的，実験内容，結果，考察，まとめまでのレポート全体を通して，読み手の立場に立って記述するよう十分に心掛けること。

付録

ロボット内部の回路図

回路図



タンサーボーグ取扱説明書 (BANDAI) P.42 より抜粋

ロボットの動作指令関数等の一覧

■ロボットの動作制御関係	補足）これらの関数は，動作の完了を待たずに戻る 動作完了を待つ必要がある場合は，Wait(0) を呼び出す（Wait 関数参照）
`Act_Fwd(int p1,int p2)`	前進動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:前進距離 [cm] 1～200 の整数値を指定（ 250 以上で無限に前進）
`Act_Bwd(int p1,int p2)`	後進動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:後進距離 [cm] 1～200 の整数値を指定（ 250 以上で無限に後進）
`Act_Turn_FR(int p1,int p2)`	右旋回 前進動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Turn_FL(int p1,int p2)`	左旋回 前進動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Turn_BR(int p1,int p2)`	右旋回 後退動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Turn_BL(int p1,int p2)`	左旋回 後退動作をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Rot_R(int p1,int p2)`	右回転動作（その場で回転）をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Rot_L(int p1,int p2)`	左回転動作（その場で回転）をさせる p1:モータ出力 最小値：1 ～ 最大値：31 の整数値を指定 p2:旋回角度 [deg] 1～360 の整数値を指定（ 400 以上で無限に旋回）
`Act_Stop(void)`	停止
■モータへの直接指令関係	
`Motor_R(int p)`	右車輪のモータ出力を直接指定（負の値で逆方向の出力） p:モータ出力 -31 ～ 31 の整数値を指定
`Motor_L(int p)`	左車輪のモータ出力を直接指定（負の値で逆方向の出力） p:モータ出力 -31 ～ 31 の整数値を指定
■時間関係	
`Wait(int t)`	引数で指定する時間待ち t: 10ms 単位で指定： 1=10ms，100=1s t=0 のときのみ特別な待ちとなる： ロボット動作関係の処理の完了を待つ
`wait_for_10ms_timer(void)`	無限ループの中に置くことで，繰り返し周期を 10ms にする タイマー割込でカウントしているため，正確な制御周期にできる
`wait_for_1ms_timer(void)`	無限ループの中に置くことで，繰り返し周期を 1ms にする タイマー割込でカウントしているため，正確な制御周期にできる
■ AD 変換関係	
`get_ad(int ch)`	AD 変換器から変換値を取得， 戻り値： int 型の変換値 ch: チャンネル番号： 0～3 の４つのチャンネルを指定可
`get_line_sensor(int ch)`	３つのラインセンサ（CdS）出力を取得（AD 変換器から） 戻り値： int 型の変換値 ch: 0～2 の整数値を指定 0=右センサ，1=中央，2=左センサ
■ PC への文字列送信関係	
`print_int( int val )`	引数で指定される int 型データを文字列に変換して，PC へシリアル送信 val: 送信する int 値を指定
`print_float(float val )`	引数で指定される float 型データを文字列に変換して，PC へシリアル送信 val: 送信する float 値を指定 （小数点以下第 2 位までを文字列に変換して送信）
`print_string( char str[] )`	引数で指定される文字列を PC へシリアル送信 str: 送信する文字列を指定 改行コードの送信の場合、"¥r¥n"と指定すると良い

RAM へのプログラム転送手順

プログラムの開発段階では，いちいちフラッシュメモリ上の ROM プログラムを総入れ替えしていると手間で時間もかかり，面倒になってくる。また，フラッシュメモリには書き込み回数の制約がある。そのため開発段階では，上手に RAM 領域（図 13 参照）を利用することが良く行われる。

搭載している MPU は RAM 領域が 2KB と小さいため，簡易的な方法をとることにする。今回の実験では，プログラムの user_main 関数を開発していくことが主な作業となる。そのため，該当するプログラムコード部分のみ RAM 上に格納することを考える。つまり，その部分のコードの変更であれば，RAM 上のプログラムを置き換えるのみで良くなる（RAM は何度も自由に書き換え可能）。

【1】user_main() 関数部分の機械語コードを RAM にダウンロード

ROM 部分のプログラムには，動作中いつでもシリアル接続経由で RAM に置くべきプログラムを受け取れるように記述してある（`int_sci3()` 割込関数や `Down_Data()` 関数で行っている）。そのため，RAM にダウンロードする際には，わざわざ**ブートモードに変更する必要はない**。

ビルド時の最終の処理段階において，RAM に置くべき機械語コードのみ抜き出す処理を行っているため，「`X 月 Y 日 ¥RobotJikken¥Release`」フォルダ内に RobotJikken.mtr というファイルも存在するはずである。これは RobotJikken.mot 機械語コードの user_main() 関数部分のみを抜き出したものである。

Step1

RAM への書き込みツール（本実験のために独自に作成したもの）がデスクトップに SysJikken.exe として置いてあるので起動する。

Step2

起動後,「プログラム選択」ボタンを押して，RobotJikken.mtr ファイルを選択する
（☞**その際，ファイルの場所が各班のフォルダであるか確認すること!!**）。

Step3

最後に「RAM へ転送開始」ボタンを押す。ロボットがコードを受信中，ロボット上の 4 連 LED が全部点灯するようにしてある。（プログラムが短い場合は一瞬全灯するのみである）

【2】ロボット上の Start ボタンを押して動作させてみる

これ以降は，コードの記述，ビルド,「RAM へ転送」の繰り返しで目的の動作となるようプログラムを開発していけば良い。ただし，RAM へ転送したコードの部分は電源を落とせば消滅する。

✎ 安定に機器を動作させる機械語コードの最終版が出来上がれば，ソースコード中の`#pragma section` の 2 行を削除（あるいはコメントアウト）してビルドして，フラッシュメモリ上の ROM プログラムの総入れ替えを行えば，すべてのプログラムコードを ROM 上に格納して動作させることができる。

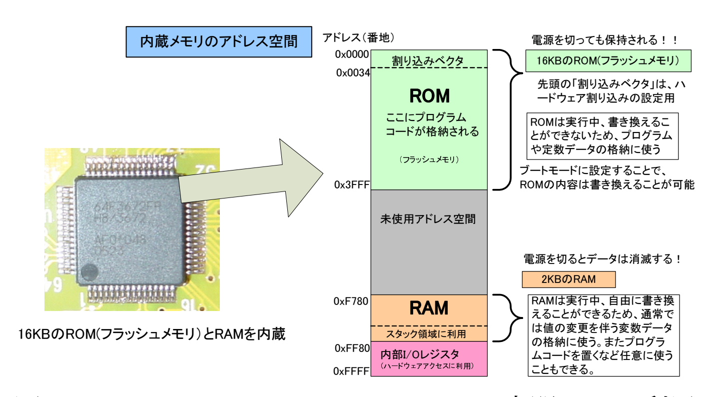


図 13: HD64F3672FP のメモリアドレス空間とその利用

while	繰り返し

書式 while(式 1)

意味 式 1 がループを継続する条件式である。式 1 を満たしている (真である) 限りループが継続する

利用例 例 1

```
i=0;
while( i< 10 ){
        (処理１)
        i++;
}
```

ループ前に i=0 が実行され，一回目の処理 1 の実行の前に条件式 i ＜ 10 が評価される。満たしていなければ，処理 1 は一度も実行されずにループは終了する。例 1 では満たしているので 1 回目の処理 1 は実行される。処理 1 の後に明示的に i++を実行しており，2 回目の処理の前に再度 i ＜ 10 が評価される。for 文との大きな違いは条件式の評価が繰り返し処理前に行なわれる点である。例 1 では結果的に処理 1 が 10 回実行される。

例 2

```
while(1){
        (処理２)
}
```

評価式である式 1 の値が常に 1 である。つまり条件式が偽となる 0 には決してなることはないため，無限にループを繰り返す。for(;;) の場合と同様にループの中断処理を行なう break 文と組み合わせて使われることが多い (break の項目参照). 組み込み機器などの制御機器では無限に動作を継続する必要があるため，while(1) が良く使われる。

for	繰り返し（ループ）

書式 for(式 1 ; 式 2 ; 式 3)

意味 ループの開始前に式 1 が実行される。式 2 はループを継続する条件式。式 3 は繰り返しごとに実行される式。

利用例 例 1

```
for(i=0; i< 10; i++){
        （処理１）
}
```

ループ前に i=0 が実行され，処理 1 が一回実行されるごとに，i++が実行される（i++は i=i+1 と同じ）。その後，条件式である i ＜ 10 が評価され，満足していれば 2 回目の処理 1 が実行される。i ＜ 10 を満たしていなければループ終了。したがって，例 1 では処理 1 が 10 回実行されてループが終了することになる。

例 2

```
for(;;){
        （処理２）
}
```

これは特殊な例であるが，式 1，式 2，式 3 を全て省略すると，無限にループを継続する。ループの中断処理を行なう break 文と組み合わせて使われることが多い (break の項目参照).

do ～ while	繰り返し

書式 do { 処理 1 }while(式 1)

意味 式 1 がループを継続する条件式である。処理 1 が実行された後に式 1 でループを継続するか評価する。満たしている (真である) 限りループが継続する

利用例 例 1

```
i=0;
do{
        (処理１)
        i++;
}while( i < 100 );
```

処理 1 がまず実行される。その後，継続して繰り返すかを条件式 i ＜ 100 で判断する。満たしていなければ，do{ } 内が繰り返し処理される。満たしていない場合はループを終了する。この例 1 では最終的に処理 1 が 99 回実行される。

break	繰り返し（ループ）を中断する

書式 break;

意味 do, for, while, switch 文を強制的に終了する

利用例 例 1

```
while(1){
   (処理１)
   if( value == 0) break;
}
```

変数 value の値が 0 であれば (if 文の使い方は次項参照), ループの条件に関わらず強制的に終了する。

continue	繰り返しの先頭に戻る

書式 continue;

意味 do, for, while の繰り返し開始位置にジャンプする。

利用例 例 1

```
while(1){
   処理１;

   if( value == 0) continue;

   処理２;
}
```

変数 value の値が 0 であれば繰り返し処理の残りの処理 2 を実行せず，次の繰り返し処理に強制的に移る。

if	条件式の評価に応じて処理コードを変える

書式　if(式 1){　(処理 1)　}else{　(処理 2)　}

意味　式 1 が評価式を満たしている（真である）場合は処理 1 が実行され，偽であれば処理 2 が実行される

利用例

例 1

```
if(i==1){
    (処理 1)
}else{
    (処理 2)
}
```

変数 i の値が 1 であれば処理 1 が実行される (==は演算子の項目参照)。そうでなければ処理 2 が実行される。

例 2

```
if(i==1){
    (処理 1)
}
```

else 以降は省略することもできる。

例 3

```
if( i>=0  && i<=5 ){
    (処理 1)
}
```

&&を用いて，i ＞=0 かつ i ＜=5 というように AND の評価を行なうこともできる。

例 4

```
if( i!=0  || i!=5 ){
    (処理 1)
}
```

|| を用いて，i!=0 または i!=5 というように OR の評価を行なうこともできる。!=は「等しくない」の意。

例 5

```
if( i<5 ){
    (処理 1)
}else if( i<10 ){
    (処理 2)
}else{
    (処理 3)
}
```

else if により複数の条件を段階的に評価できる。i ＜ 5 を満たしていれば処理 1 が，満たしていなければ else if により i ＜ 10 を評価し，それを満たしていれば処理 2 を実行する。どれも満たしていない場合に処理 3 が実行される。

switch, case	条件式の評価に応じて処理コードを変える

書式

```
switch( 式１ ){
  case 定数１:
    処理１
    (break;)
  case 定数２:
    処理２
    (break;)
  default:
    処理３
}
```

意味 式 1 と一致する定数があれば，その処理が実行される。一致するものがなければ default の部分の処理が実行される。ある case 文の処理を行なった後，それ以降の case 文の処理判定が必要なければ break で switch 文から抜けることができる。尚，式 1 は整数値になる式，定数の部分も整数値である必要がある。同様の処理は if 文を用いても記述することができるが，処理をある値の定数で分岐させたい場合には，switch 文の方が分かりやすく記述できる。

利用例 例 1

```
switch( value ){
   case 0:
     処理１;
     break;
   case 1:
     処理２;
     break;
   case 2:
     処理３
     break;
   default:
     処理４;
}
```

変数 value の値が 0 であれば処理 1，1 であれば処理 2，2 であれば処理 3，その他は処理 4 が実行されて switch 文を終了する。

goto	ジャンプする

書式 goto ラベル;

意味 ラベルを定義した個所にジャンプする。goto 文はとても便利な分岐方法であるが，むやみに使うとプログラムの流れを追跡しにくくなる。ほかに方法がない場合や特別な理由がない限り使わないようにすること。

利用例 例 1

```
my_func(){

   if(i == 1) goto label1;

   i=i+3;

label1:

   i=j+2;
}
```

例のようにラベルの定義にはコロンを使う。最初の if 文が真であれば label1:を定義した部分にジャンプする。switch や continue, 関数から戻る return，関数の定義，{} を活用すれば goto を使う必要はほとんどない。なるべく goto 文は使わない。

演算子

==	値が等しいかの評価を行なう。
！=	不等評価を行なう。
＜	大小関係の判定（小なり）
＞	大小関係の判定（大なり）
＜=	大小関係の判定（以下）
＞=	大小関係の判定（以上）

書式　式 1 == 式 2，式 1 <= 式 2
意味　2 つの式あるいは値を評価して，真であれば True, 偽であれば False を返す。if 文や while 文等の条件式の中で使われる
利用例　例 1

```
if( i==1 ) i=i+1;
if( j<=1 ) j=j+1;
if( k>=1 ) k=k+1;
if( m!=1 ) m=m+1;
if( n <1 ) n=n+2;
```

注意 間違えやすい例

```
if( i=1 ) break;
```

==とするところを=と間違えて記述してもコンパイル時にエラーにならない。この場合には i=1 の代入式として処理される。i=1 の値は真である（０でない）ため break 文が実行される。

++	インクリメント演算子
- -	インクリメント演算子

書式　変数++, 変数- -, ++変数，- -変数
意味　変数に 1 加える（＋＋の場合），1 減じる（- -の場合）
利用例　例 1

```
i=0; j=0;
i++;
j--;
val=++i;
val=i++;
```

++(- -) を変数の前に置くか，後に置くかで処理される順序が異なる。val=++i; の場合には，変数 val に i の値を代入する前に i=i+1 が実行される。val=i++; の場合には i の値を val に代入してから i=i+1 が行なわれる。

論理演算子

&&	論理 AND 演算子
||	論理 OR 演算子

書式　式 1 && 式 2 , 式 1 || 式 2

意味　&&: 式 1 と式 2 ともに真のときに真となる。|| : 式 1 と式 2 のどちらかが真であれば真となる。

利用例　例 1

```
if( a==1 && b==1 && c==1) i=10;
if( x==1 || y==1 || z==1 ) i=0;
```

複数の条件式の値を評価することができる。

ビット演算子

&	ビットごとの AND 演算子
|	ビットごとの OR 演算子
^	ビットごとの排他的 OR 演算子
~	ビットごとの反転
<<	左シフト演算子
>>	右シフト演算子

書式　例： 式 1 & 式 2 , 式 1 | 式 2

意味　式の値をビットごとに演算する

利用例　例 1

```
a1=0x00ff;
a2=0xff00;
a3= a1 & a2; // a3=0x0000 となる
a4= a1 | a2; // a4=0xffff となる
```

2 つの変数 a1 と a2 をビットごとに AND 演算，OR 演算している例。

例 2

```
a1=0x01;
a2=0x10;
a3= a1<<1; // a3=0x02 となる
a4= a2>>1; // a4=0x08 となる
```

2 つの変数 a1 と a2 をビットシフトしている例である。

関数（以降の関数は本実験では使う必要はない）

rand()	0〜32767 の間の擬似乱数を生成する
srand()	乱数ジェネレータを初期化する

書式 rand(), srand(seed_value)

意味 擬似乱数の生成，seed_value は乱数の生成に使われる初期値

利用例 例 1

```
value=rand();
```

0〜32767 の間の乱数が生成され value に代入される。尚，いきなり rand() によって乱数を生成すると，毎回毎回同じ乱数値が生成される。そのため rand() を使う前に srand() を一度だけ呼び出して乱数系列のシード値を設定する必要がある。

例 2

```
srand( time(NULL) ); //srand はプログラムの最初で一回だけ呼び出せば良い。

value=rand();
```

この例はシード値を現在の時間から決めることで，毎回同じ乱数が生成されないようにした簡単な例である。

printf()	文字列を出力する

書式 printf(”.....”,....)

意味 printf: 文字列を出力する。

利用例 関数の機能は豊富であるため，ここでは簡単な利用例のみ書いておく。詳しくは専門書を参考にして下さい

例 1

```
printf("My Name is CIST.\n");
```

”.....”で囲んだ文字列を出力する。最後の”\ n”は「改行しなさい」という特別な意味を持つ。

例 2

```
i=100;
printf("Answer= %d \n", i);
```

出力する文字列の部分に`%d`が挿入されている。「整数値 i の値を`%d`の部分に代入して表示しなさい」という意味となる。

例 3

```
value=0.1234;
printf("Answer= %f \n", value);
```

出力する文字列の部分に`%f`が挿入されている。「小数点数 value の値を`%f`の部分に代入して表示しなさい」という意味となる。

例 4

```
i=10;
value=0.1234;
printf("Answer= %d , Value = %f \n", i, value);
```

この例のように複数の代入をすることが可能である。この場合には「`Answer = 10 , Value = 0.123400`」のようにそれぞれ順に代入されて出力される。